MediateSpace: Decentralised Contextual Mediation Using Tuple Spaces

Danny Matthews
School of Informatics
University of Sussex
Brighton, BN1 9QH, UK
d.matthews@sussex.ac.uk

Dan Chalmers
School of Informatics
University of Sussex
Brighton, BN1 9QH, UK
d.chalmers@sussex.ac.uk

Ian Wakeman
School of Informatics
University of Sussex
Brighton, BN1 9QH, UK
ianw@sussex.ac.uk

ABSTRACT

With almost four billion sensor-equipped mobile devices on the planet, the way is open for a variety of new context-based applications and services. However, this new opportunity creates concerns over *privacy and access control* and necessitates a *robust and scalable solution*. We propose the MediateSpace middleware which is a *decentralised tuple space with contextual mediation capabilities for both data distributors and consumers*. Distributors may restrict access by requiring the satisfaction of a *contextual condition* and consumers may restrict which data enters their computer (*tuple conditions*). *Distributed X-Trees* (a development of *R-Trees*) are used to achieve decentralisation.

The system also provides *Restricted Context Sharing*, *Triggers* (performing actions upon matching certain conditions or data patterns), *Module Handlers* (simplifying the processing of received messages), *Context Scripting* (allowing the dynamic addition, removal or augmentation of structures such as triggers) and *State Management* (allowing state to be read and stored semi-persistently).

MediateSpace could be used to support a myriad of possible applications such as context dependent data collection, collaborative tools for geographically co-located individuals and context-aware file-sharing.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.3.2 [**Programming Languages**]: Language Classifications—*Very high-level languages*

General Terms

Algorithms, Design, Languages

Keywords

Context awareness, Contextual mediation, Middleware, Decentralised, Tuple space, Mobile ad hoc networks

Middleware for Pervasive Mobile and Embedded Computing 2011 Lisbon, Portugal

1. INTRODUCTION

The proliferation of sensor equipped devices has increased tremendously in recent years, with estimates suggesting almost four billion worldwide [1]. People are becoming ever more comfortable with the use of sensors in everyday life (e.g. GPS and accelerometers); with every reason to expect this familiarisation to grow as time goes by. This growth in availability and public understanding opens the way for a variety of new applications and services, but at the same time creates new concerns over *privacy* and *access control*.

By developing previous work in the area (§3), we aim to support this growth through the development of a *fully decentralised and scalable distribution framework* suitable for supplying data and services to end users, whilst also enabling privacy and access control through the specification of powerful *contextual conditions*, *information filtering* and other structures (§4). These structures offer highly granular privacy and access control not achievable through other means (e.g. encryption). *Full decentralisation* (§5) ensures *fault tolerance* and *scalability*.

2. A MOTIVATING EXAMPLE

A variety of possible applications can be supported using MediateSpace. As an example, we consider a context dependent data collection platform which can be extended and updated on-the-fly without requiring a restart of the application.

- The middleware ensures that all sensor-carrying devices are capable of participating in data collection by having each sensor implementation adhere to an appropriate common interface. For example, a Location context could be implemented using GPS, beacons, or any other means. If a device does not have access to a required type of context they can request it from a nearby participant.
- Triggers may be defined to collect data upon particular conditions occurring. For example, a researcher may only be interested in obtaining data from particular locations within particular periods of the day. Filters may be used by end-users to restrict the data they receive.
- The middleware is not restricted to the distribution of sensor data, accepting payloads of arbitrary structure. This could be useful for distributing project information or software updates to end-users. The integration of these files is made straightforward by allowing the researcher to specify module handlers which stipulate how each file should be handled upon receipt.
- Finally, the data collection platform can be altered to take into account changing requirements. For instance, a researcher may wish to start collecting data for additional types of context, or may wish to change the conditions for receiving data.

This may be achieved by using the context scripting capabilities of the middleware to insert the additional contexts and alter the existing triggers.

3. RELATED WORK

Our work is heavily based on previous research into Tuple Spaces [2–7], Contextual Frameworks [8–14] and Decentralised Communication. This section discusses these works in detail.

3.1 Tuple Spaces

A tuple space is a shared space accessible to many entities [2]. These entities may be separate processes or even separate machines communicating with the space over a network. Tuple spaces store tuples - which are simply packages holding arbitrary data with a pattern which must be matched in order to retrieve it. A tuple space has three basic operations which can be performed on it:

- `out` - Place a new tuple in the space.
- `rd` - Read a copy of a tuple from the space.
- `in` - Read and remove a tuple from the space.

The `rd` and `in` commands are blocking operations, meaning that they will not return until a matching tuple is found. There are also non-blocking variations of these commands (`rdp` and `inp`) and versions for matching multiple tuples at a time (`rdg` and `ing`) [3].

Tuple spaces have shown themselves to be light-weight structures with good utility [2–4], having the major advantage of supporting asynchronous access via unreliable/temporary channels which ensures that data remains accessible even when the source becomes unavailable. This makes it an appropriate choice for supporting fully decentralised systems within a mixed-capability environment.

3.2 Distributed Tuple Spaces

One of the key decisions to be made when designing a distributed tuple space is the choice of data structure used to represent it.

For example, the Peer-to-Peer Tuple Space middleware [4] is designed to handle distributed resource brokering. All resource requests and resource usage information is shared via a tuple space which is distributed through the use of a distributed hash table (DHT) and the publish-subscribe pattern.

Other implementations take a different approach. For instance, Linda In a Mobile Environment (LIME) [3] and PeerWare [5] achieve the illusion of shared memory through the use of the Global Virtual Data Structure (GVDS) [6]. This is a global data structure which may be divided across the network nodes and then merged when nodes come into communication distance of one another.

LIME uses the GVDS to create a virtual unified tuple space of all nodes within wireless communication distance, making it straightforward to ensure that all updates, additions and removals are reflected in the global tuple space. LIME also supports "reactions" which simplify the task of reacting to changes in context by observing tuples as they enter the system and triggering an action whenever they match some given set of properties. Reaction rules are a powerful concept which can be used to support many tasks such as data replication [15].

PeerWare uses the GVDS to represent two main types of entity: nodes and documents (the payload). Sharing of the GVDS is achieved through the use of a peer-to-peer network.

The GVDS operates very successfully in circumstances where nodes are geographically proximate to one another, but suffers in widely distributed networks because of the low likelihood of nodes coming into communication range of one another. In addition, both the LIME and PeerWare systems do not provide facilities for contextual mediation which limits their data filtering capabilities.

The TinyLIME middleware [7] extends LIME and focuses on retrieving and aggregating sensor values from wireless motes. By using the distributed shared memory approach of LIME, it facilitates the sharing of sensor values over the network. This approach is most appropriate when sensors are sparse or isolated. In addition, TinyLIME reactions have been enhanced to include contextual conditions (e.g. triggering only if the temperature is between 20° and 30°) which gives it superior filtering capabilities over LIME, but still suffers from a lack of expressivity.

3.3 Contextual Frameworks

A contextual framework is software which attempts to reduce the burden on context-aware application design by abstracting the sensor reading and evaluation processes via the use of small APIs and context models. They have gradually increased in sophistication; from providing only simple abstraction and inference mechanisms (such as signalling when an individual leaves a room) [8] to providing more powerful inference mechanisms [9–11] through specialised languages (e.g. OWL) and conditional structures (e.g. $\wedge$, $\vee$, $\exists$).

Many of these systems have been designed to be either centralised or localised (requiring direct communication with a server for data processing), and for small-scale or localised networks these systems may be sufficient. However, they are largely inadequate for large scale networks. Some researchers have taken this into account in their designs with varying degrees of sophistication. Some simply provide the sharing of sensor data between devices via short-range communication (e.g. BlueTooth) [12], whilst others combine this with infrastructure to support more powerful applications [13].

The frameworks typically offer good heterogenous interoperability (between devices, programming languages etc) through their use of XML based communication (e.g. RDF/OWL, SOAP).

3.3.1 Localised Frameworks

The Context Toolkit [8] was one of the first context frameworks. It separated the application logic from the sensor implementation by encapsulating the sensor reading code within a widget containing various "attributes" (such as last sensor value) and callback functions which trigger on high-level events occurring (such as an individual leaving a room). However, it lacks any native facility for forming complex conditionals and makes extension difficult as each ontology is tightly coupled with the sensor reading code.

The CoBra ontology system [9] and The Context Management Framework (CMF) [10] both provide much more sophisticated ontologies, allowing higher levels of context to be inferred. CoBra is defined using OWL which allows for far richer semantic information about entities to be expressed which allow more powerful inferences to be made. CMF allows the specification of higher level ontologies by combining the values of two or more lower level ontologies. For example, several sound based ontologies (`Harmonicity`, `SpectralSpread`, `Transients`) could be combined to form a higher level `SoundType` context (`Car`, `Elevator`, `RockMusic`). The model also accounts for imperfect or partially ambiguous sensor data by allowing for ontologies to be modelled using fuzzy sets and a Bayes probability model and provides quite sophisticated condition evaluation facilities (AND, OR and NOT logical connectives).

SOCAM [11] provides excellent evaluative capabilities, allowing the use of logical connectives (e.g. $\wedge$, $\vee$, $\neg$), quantifiers (e.g. $\exists$) and all of the capabilities provided by RDF/OWL.

Although CoBra, CMF and SOCAM all provide greatly improved context evaluation capabilities, their localised nature prevents them from effectively supporting wide-area information distribution applications.

3.3.2 Wide-Area Frameworks

One of the first decentralised frameworks was Hydrogen [12], where devices communicated via short range protocols such as BlueTooth to share sensor readings. This was an important first step but lacks a mechanism for flexible context evaluation and extendability. More recently researchers have been looking into the feasibility of wide-area sensor frameworks using a combination of server machines with pocket switch and related ad-hoc networking solutions. For example, Santa and Gómez-Skarmeta [13] provided car users with useful information through the combination of Vehicular Ad-Hoc and cellular networks. This approach is undoubtedly promising but lacks generality in the applications supported and provides limited context filtering.

Eisenman et al. [14] have gone further, ambitiously proposing the addition of a new wireless sensor edge for the current Internet to support large scale sensing. They emphasise three key principles:

Network Symbiosis The harnessing of existing knowledge and technologies.

Asymmetric Design Acknowledging device asymmetry to provide a better service. For example, computationally intensive tasks could be pushed onto more capable nodes.

Localised Interaction The communication range of all network nodes should be heavily constrained within "spheres of interaction" - with the motivation being a simplified design and improved communications performance. The limitations of this design are overcome through opportunistic delegation, which is the process of delegating jobs to nodes as they are encountered. These nodes may themselves delegate and so on. This allows nodes to communicate across great distances while still retaining the benefits of localised interaction.

Attempts have been made to incorporate each of these principles into the MediateSpace design.

3.4 Decentralised Communication

Centralised communication between nodes in a network is conceptually simple as all communication passes through a central server. Fully decentralised communication does not have this luxury and is hence a more difficult problem to solve.

Solutions vary depending on the dimensionality of the data being communicated. If the data is one-dimensional then a *Distributed Hash Table (DHT)* may be appropriate. However, if the data is multi-dimensional more complicated structures are needed. One solution is to incorporate a space-filling curve such as the *Hilbert curve* into the DHT, which allows the production of one-dimensional numeric objects from multi-dimensional data whilst retaining the "distance" between the objects (where distance may refer to many things such as geographical distance, degree of similarity etc).

Alternatively, more powerful tree based structures may be used which allow sophisticated topological queries to be applied to multi-dimensional data. Probably the best understood of these structures is the *R-Tree* [16] which allows the efficient lookup and storage of geometric data. However, it tends to suffer a degradation in performance as the dimensionality of the data increases [17].

The *TV-Tree* [18] attempts to rectify this deficiency by reducing the number of dimensions which need to be analysed. This is achieved by excluding from consideration any dimensions upon which the objects have identical values.

The *X-Tree* [17] takes an alternative approach which is based on the observation that the main cause of degradation in multi-dimensional *R-Trees* is a high level of overlap between objects, which makes it necessary to traverse large proportions of the tree during lookup. This resulted in a structure with an emphasis on reducing overlap and increasing the spatial locality between nodes for which a high overlap was unavoidable. Both of these alternative structures exhibit improvements over the original *R-Tree* with the *X-Tree* proving itself to be a very significant improvement.

4. MEDIATESPACE

We now present MediateSpace which is a decentralised middleware solution primarily designed to allow contextually-aware information distribution. That is, information can only be delivered to a client who passes the associated contextual conditions. The system workflow is summarised in Figure 1.

4.1 Overview

We based our work on the *Tuple space paradigm* [2], with all system structures and payload data being stored in tuples. We extended the paradigm with notions of context and decentralisation.

The tuple space has been subdivided into three *Sub Tuple Spaces* (see Figure 2a). The conditional space is responsible for messages (§4.2), the caching space for context exchange (§5.2) and the communication space for message exchange (§5.3).

The system also supports a number of other operations designed to provide additional information filtering capabilities and to ease the development of context-aware applications:

Context Definition (§4.5) Contexts are defined in two parts: The context protocol (interface) and it's implementation for a particular sensor. This allows multiple implementations to be written for a context using multiple sensors.

Context Exchange (§5.2) Participants may request context values from others within the network. This would be used when attempting to satisfy a condition for which the participant does not have the necessary sensor or context implementation.

Triggers (§4.6) Triggers perform an action whenever the specified event is satisfied. Events may be either a contextual condition or a condition concerning tuple structure.

Filters (§4.7) Filters can be used to apply contextual restrictions on which *parts* of a tuple may enter the system. For example, a participant may refuse entry to executable files.

Module Handlers (§4.8) Module Handlers provide a means of dealing with messages as they enter the system. Each handler specifies a pattern (and optionally additional conditions) which messages are compared against. When a message matches, the driver (a small program) associated with the handler is executed on the matching portion of the message.

Context Scripting and State Management (§4.9) Context scripts allow participants to modify the system state by adding, removing or augmenting structures. For example, adding or deactivating a trigger.

State management makes it possible to retain and restore state semi-persistently by including the appropriate statements within a message tuple.

All operations are defined using a MediateSpace definition language which is tailored to the design goals of the middleware and is based in part on the works discussed above (§3). Only the most important subset of the language is shown.

A location-based game called Hunt is described in all examples.

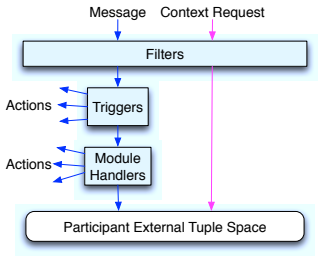


Figure 1: System Workflow

4.2 The Message Tuple

The message tuple contains the information which is distributed across the network (see Figure 2b). It is composed of a contextual condition (§4.3), an *Advert* (used by participants to decide whether a message is applicable) and one or more payload modules which contain the data of the tuple. Several standard modules have been defined but modules of any structure can be specified.

4.3 The Contextual Language

The contextual language (see Figure 3a) makes use of predicate logic to make it as flexible as possible. Specifically, it supports universal and existential quantification (∀, ∃), conjunction, disjunction, exclusive or and negation.

Conditions may be connected together using two types of command. The first simply binds commands together using logical connectives. The second type is much more powerful as it allows you to either express that all conditions must be true (∀) or that O are true, where $n \leq O \leq m$ (∃ n .. m). Curly braces are used to separate blocks of conditions so that they may be connected together into increasingly complex statements.

Conditions can be made more flexible by using wildcards within their parameters. Wildcards are specified using the "*" character (see Figure 3a for an example).

4.4 The Tuple Language

The tuple language allows evaluations to be performed on sections of a received message tuple. This allows users to apply mediation to the data of each tuple they receive; giving fine-grained control over how it is handled. Message tuples are represented internally as XML documents which the language queries (in a similar fashion to XSLT). The language supports blocks and all of the predicates supported by the contextual language.

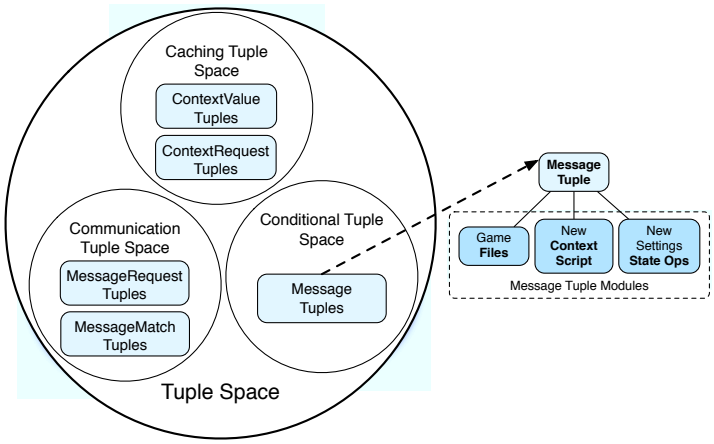
Figure 3b shows an example. The `*` operator indicates that the right operand is of a structure which could be repeated many times. For instance, the `sf:file` structure is repeated for each file in the tuple. Conditions which follow within square brackets should be executed for each of these structures. The example `TupleCondition` states that the file should be of type HGF or HEF and that the tuple should be associated with the *Hunt* game.

4.5 Defining Context

In order to reason about contexts it is necessary to:

1. Create a model of the context.
2. Map the values from an appropriate sensor onto the model.

The former is achieved by defining a `Context` structure while the latter requires definition of a `ConcreteContext`. `Contexts` and `ConcreteContexts` are analogous to object-oriented interfaces and classes respectively. `Contexts` allow the definition of a list of abstract method signatures and an ontology of possible values



(a) The MediateSpace Tuple Space

```
MessageTuple {
  Meta {((tupleID, "update-8"),(appID, "hunt"))};
  Condition { contextualCondition };

  Advert {
    (Files, ["Hunt Update", "Graphical Update"],
            ["Hunt Update", "Engine Update"]),
    (ContextScript, ["Location Event", "Pudding Lane"],
                    ["Location Event", "Big Ben"]),
    (StateOps, ["Version Number Updated"]);
  }

  PayLoadModules {
    StateOps { state_out(1.03, versionNum) };
    ContextScript { ADD { TRIGGER {locPud, locBen}}};
    Files {((graphic-update, "gup.HGF"),
            (engine-update,  "eup.HEF"))};
  }
}
```

(b) The MessageTuple

Figure 2: The Tuple Space and Example MessageTuple

and relationships. `ConcreteContexts` implement the methods and ontology with regard to a particular sensor.

`Contexts` are able to inherit from other `Contexts` and `ConcreteContexts` are able to extend another `ConcreteContext` (single inheritance) and may be declared abstract to allow partial implementation when one or more contracts cannot be fulfilled; serving to reduce code duplication and memory footprint.

The system supports six primitive (`Boolean`, `String`, `Integer`, `Float`, `Date`, `Ontology`) and three complex types (`AssociativeMap`, `Pair`, `List`). The `Ontology` type accepts any value defined as part of the ontology for the current `Context`.

Each defined method must either be handled by a driver or be defined as a `SimpleContract`. Simple Contracts specify a constant value to always be returned when the method is called.

It is sometimes necessary to obtain data from another `Context` to formulate a result. In these cases, `Dependencies` may be specified as part of the `Context`. `Capabilities` and `Properties` of a context may also be defined.

Figure 4 defines the `HuntLocation` context whereby each location is represented by GPS coordinates. It also illustrates how `ConcreteContexts` may declare allowable levels of deviation from the values declared for the ontology by using the ~ symbol. For instance, the example shows that `HuntLocationGPS` will regard the user as being at `BIG_BEN` provided the sensor reading does not deviate by more than 0.000030 in any direction.

```
EXAMPLE (USING QUANTIFICATION):

  { forall HuntLocation.HuntLocation(BIG_BEN),
    Std.Compare(Time.getCurrentTime(), ">=", **:00),
    Std.Compare(Time.getCurrentTime(), "<=", **:01) }

EXAMPLE (USING LOGICAL CONNECTIVES):

  { HuntLocation.HuntLocation(BIG_BEN) &&
    Std.Compare(Time.getCurrentTime(), ">=", **:00) &&
    Std.Compare(Time.getCurrentTime(), "<=", **:01) }
```

(a) Example Contextual Conditions

```
namespace ms mediateSpace
namespace sf simpleFile

ms:meta
  ms:metaEntry
    ms:metaName = "appID"
    ms:metaValue = "hunt"
  ms:metaEntry (...)
sf:files
  sf:file fileType = "HGF" (...)
  sf:file fileType = "HEF" (...)

TupleCondition {
  {
    sf:files * file [
      fileType == "HGF" || fileType == "HEF"
    ]
    &&
    ms:meta * metaEntry [
      metaName == "appID" && metaValue = "hunt"
    ]
  }
}
```

(b) An Example Tuple Condition & Tuple Fragment

Figure 3: Supported Conditions

4.6 Triggers

Triggers monitor the device's state and the contents of the tuple space in an effort to match a condition. If this condition is successfully matched, an action is performed which typically involves injecting a tuple into the tuple space. Triggers can use both contextual and tuple conditions, and these conditions may be nested to any depth and in any order.

It is often desirable that an action be performed both when the condition has been met and when it has not. For instance, switching off a light when a user leaves a room and switching it on when they enter. This use case has been handled simply by allowing a qualifier to be appended to the condition (`onSuccess` or `onFailure`).

In addition there are cases where the temporal relationships between contextual events are important. An example could be determining if a room has been empty for a given period so that devices such as CD players can be switched off. This is catered for by the `ConditionSequence` construct which allows the specification of multiple conditions grouped by the temporal relationships which should exist between them. Triggers may operate in two modes:

CONTEXT Mode The conditions specified will all be in reference to the context of the user's device. For example, a trigger may fire when a particular temperature is reached. In this mode tuple conditions are not applicable.

TUPLE Mode Evaluation occurs whenever a tuple enters the system. Both contextual and tuple conditions are applicable.

```
Context HuntLocation {
  Contracts { BOOL HuntLocation(ONTVAL loc); }
  Ontology { BIG_BEN, PUDDING_LANE, HMS_BELFAST; }
  Capabilities { location; }
}

ConcreteContext HuntLocationGPS {
  Meta { contextDriver="HuntLocGPS", quality=5; }
  Ontology {
    (PUDDING_LANE, [(==, [51.303652, -0.5720]),
                    (~,  [0.000050, 0.000050])]),
    (BIG_BEN,      [(==, [51.30290, -0.72848]),
                    (~,  [0.000030, 0.000030])]),
    (HMS_BELFAST,  [(==, [51.302368, -0.45300]),
                    (~,  [0.000050, 0.000050])]];
  }
  Properties { ((location, [GPS])); }
}
```

Figure 4: Contexts and Concrete Contexts : An Example

4.7 Filters

Filters are used to reject unwanted interaction through the use of the contextual and tuple conditions discussed above. For example, a device may wish to only work with a particular named tuple such as "hunt-update", or may wish to restrict access to their sensors. The flexibility of the tuple condition structure makes it possible to have full control over which data enters the user's device. This includes the ability to restrict context structure changes and state operations (§4.9). As with triggers, filters can also be used in two modes:

TUPLE Mode The filter will be evaluated each time a tuple enters the system. Each filter specifies a pattern which is matched against the XML representation of a message on receipt. This denotes the part of the message to which this filter corresponds. For example, if a filter were applied to the `sf:files` structure in Figure 3b and the filter were to fail, this whole structure (i.e. all files) would not be accepted.

SENSOR Mode The filter will be evaluated each time an external device requests sensor data from the user's device. In this mode any contextual conditions refer to the context of the requesting entity. For example, the requester may be required to be in the same location as the user's device.

4.8 Module Handlers

When a message is received, the enclosed payload modules must be handled appropriately. To this end, Module Handlers can be defined. Similarly to filters, each handler specifies a pattern to be matched to part of the message, along with an optional filtering condition. When a match is found, the appropriate driver is executed with the matching node as parameter.

Each **Matches** clause is executed in turn until a match is found. The example in Figure 5 processes the *Hunt* game updates.

4.9 Context Scripting and State Management

Context scripts allow tuples to modify and augment the system structures (see Figure 6a). An example of use can be seen in Figure 2b where two new trigger events are added to the *Hunt* game.

State management makes it possible to retain state for an arbitrary period of time. For instance, consider the task of building an application which reduces the CD player volume when the phone is in use, and then reverts the volume once the call has ended. The volume must be stored persistently for some period before it is re-instated. Parameters can be specified which affect the period of time the data is retained and the structures used for retention (see Figure 6b).

```
ModuleHandlers {
  Matches sf : files * file {
    Meta {
      ((driver, "engine-update"),
       (handlerName, "An Engine Update."))
    }
    TupleCondition {
      {
        fileType == "HEF" && ms:meta * metaEntry [
          metaName == "appID" && metaValue == "hunt"
        ]
      }
    }
  }
  Matches sf : files * file {
    Meta {
      ((driver, "graphics-update"),
       (handlerName, "A Graphical Update."))
    }
    TupleCondition {
      {
        fileType == "HGF" && ms:meta * metaEntry [
          metaName == "appID" && metaValue == "hunt"
        ]
      }
    }
  }
}
```

Figure 5: Example Module Handlers

```
action   = ADD | REMOVE | ACTIVATE | DEACTIVATE | UPDATE
modifier =  TRIGGER | FILTER | CONTEXT |
            CAPABILITY | PROPERTY | CONCRETE_CONTEXT
```

(a) Context Scripting Commands

```
// Data structure of the given type and name.
state_create(stateType, TTL, var)

state_out(value, var)

// Optional index into the data structure.
state_in(var, index)
state_rd(var, index)

stateType = QUEUE | STACK | LIST | VARIABLE
```

(b) State Management Commands

Figure 6: System State Manipulation Commands

5. DECENTRALISED OPERATION

MediateSpace is designed to be completely decentralised which is important as it helps to ensure scalability. This is of utmost importance due to the huge number of nodes which could potentially make use of the system (as discussed in §1). All communication between nodes is represented as XML which has the benefit of allowing the system to be language-agnostic.

There are four main issues to consider during design:

1. How do nodes communicate with one another?
2. How do nodes gain access to the `Contexts` of other nodes?
3. How do nodes query the system for appropriate messages?
4. How do you ensure both scalability and fault-tolerance?

5.1 Network Structure

The network recognises two distinct types of node:

- Regional Nodes
- Participant Nodes

This distinction was made to enhance scalability and to reduce the level of complexity imposed on the majority of nodes.

Regional nodes maintain links with multiple geographically proximate Participants and are responsible for communicating with the other Regional nodes in the network. They house most of the networking complexity, with Participants only being required to bind to a single regional node (and no other Participants). It is intended that the more capable nodes take the Regional role. This many-to-one design aids scalability as it dramatically reduces the number of connections necessary to ensure full connectivity.

Both types of node have access to two tuple spaces: *internal* and *external*. However, the functionality of these spaces differ depending on the type of node. Figure 7 summarises these spaces.

5.1.1 Participant Nodes

Participants represent those users who simply wish to send/receive messages and contextual information. They communicate via the most geographically proximate regional node which they bind to when entering the network. The semantics of the participant tuple spaces are now discussed:

Internal Tuple Space This is stored on the local disk and is used to store and retrieve all of the *Contexts*, *ConcreteContexts*, *Triggers* and other system structures available to the node.

External Tuple Space The external space is also stored on the local disk but has more complicated write semantics. Writing to the space has the effect of making the tuple available to the regional node to which this node is bound. These write semantics are handled by the regional node.

5.1.2 Regional Nodes

Regional nodes are responsible for maintaining a region of the network and for communicating with other regions. Specifically, they are responsible for maintaining the part of the network which resides over a given geographical area and for distributing and receiving tuples which are of relevance to their bound participant nodes. The semantics of the regional tuple spaces are now discussed:

Internal Tuple Space Populated with all of the tuples present within the external tuple spaces of the bound participant nodes. This is a logical rather than physical tuple space, i.e. rather than storing the tuples locally, the space is dynamically constructed as the union of every bound tuple space.

External Tuple Space Regional nodes can interact with this space using a simple interface. However, the space semantics are actually extremely complex and involve the efficient routing of tuples across the network.

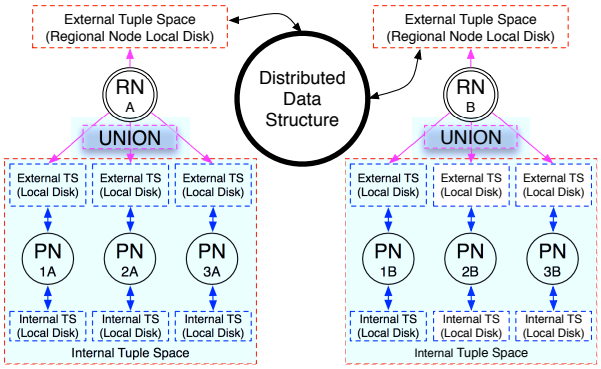


Figure 7: The Distributed Tuple Space

5.2 Exchanging Context

There will be circumstances where a participant cannot access the types of context necessary to receive a message. In these circumstances, participants can request information from another participant via the exchange of `ContextRequest` and `ContextValue` tuples. However, care must be taken because contexts may only be applicable under certain conditions. For example, a Location context may only be valid if the participant is less than 25 feet away.

Each regional node is aware of it's position within geographical space and can derive the positions of other regional nodes by querying the distributed data structure. These properties make it straightforward to restrict access by distance. Regional nodes may continue issuing requests iteratively until all further regions lie beyond the maximum distance from the original requesting node. Other restrictions can be specified using filters (§4.7).

5.3 Exchanging Messages

To restrict the search space for messages, query conditions are obtained from three sources. All messages accepted under these conditions are candidate messages:

Positive and Negative Conditions Defining conditions which are always or never of interest respectively.

Contextual Filters Messages matching filter conditions are not accepted. Hence, they are another form of negative condition.

Available Contexts Contexts which have already been evaluated and are available to the participant.

The process of querying the system for messages is undertaken in several steps and involves the `MessageMatch` tuple transitioning sequentially through each of five states.

We first consider the process with the simplification that participants have direct access to one another. We then modify the discussion to take into account the distributed nature of the system.

5.3.1 Simplified Exchange

Requesting Context A participant issues a `MessageRequest` tuple to their external space. Each message provider inspects it and collates a list of messages which are potentially within the given conditions (candidates).

Match Refinement The above matching step does not take condition parameters into account when collating candidates (with the exception of `Ontology` parameters). Parameters must be analysed because their use can alter the semantics of a condition. For example, the conditions `Achieved("BigBen")` and `Achieved("PuddingLaneFire")` refer to different achievements.

Each condition is compared against the `MessageRequest` and the remaining contexts for each message are placed into the `MessageMatch` tuple.

Context Available The participant collects as much of the requested context as possible and places it into the `MessageMatch` tuple. If the context is available locally, it is simply read and added. Otherwise `ContextRequest` tuples are issued to geographically proximate nodes which then work to obtain the required context (§5.2).

Message Stubs Available When the `MessageMatch` tuple is received by the message supplier, each message is evaluated using the supplied context. For each message that a participant satisfies, a message stub is created and placed into the `MessageMatch` tuple. Message stubs consist of the `Advert` component of the `Message` tuple (§4.2).

Message Stubs Filtered Each message stub is evaluated against any filters specified on the participant, with those stubs (or parts of stubs) satisfying the conditions being removed from the `MessageMatch` tuple.

Concrete Messages Available When the `MessageMatch` tuple is returned to the message supplier, the remaining stubs are replaced with the parts of the messages to which they correspond and are made available to the participant.

5.3.2 Distributed Exchange

Definition Let PN **NX** represent a Participant Node which is bound to Regional Node RN **X**, where $\mathbf{N} \in \mathbb{N}$ and $\mathbf{X} \in \{A, B, .., Z\}$.

1. PN 1A writes a `MessageRequest` tuple to it's external space.
2. RN A reads the `MessageRequest` tuple from it's internal space and:
   (a) Writes the tuple to its' own external space (resulting in the tuple being routed to the external space of RN B).
   (b) Collates a list of candidate messages from it's own internal space (the external spaces of PN 2A and 3A), which are inserted into a `MessageMatch` tuple and written to its' own internal space.
3. RN B reads the `MessageRequest` tuple from it's external space and collates a list of candidate messages from it's internal space (the external spaces of PN 2B and 3B) which are inserted into a `MessageMatch` tuple and written into the external space of RN B (resulting in the tuple being routed to the external space of RN A).
4. RN A reads the `MessageMatch` tuple from it's external space.

RN A and B will now be able to communicate directly by obtaining one another's address from the `MessageMatch` tuple. Direct communication continues until the exchange is complete.

5.4 The Distributed Data Structure

MediateSpace uses a Distributed X-Tree [17] for communication between regional nodes. X-Trees (a development of R-Trees [16]) were designed to handle geometric data (points, surfaces etc) of arbitrary dimensions so lend themselves well to the types of network operations which need to be performed.

X-Trees simplify network setup and maintenance as they are well-equipped for representing geographical regions and the association of each regional node with k proximate nodes is straightforward as nearest neighbour algorithms are well known.

Contextual conditions are used as an index for message storage because of their use as search terms during message lookup. Since each message usually requires the satisfaction of multiple conditions this index must be multi-dimensional. Each condition can be mapped to a dimension with some total ordering. This mapping may be over the values of an ontology or some other range of values.

For example, a 2-dimensional index would be required to represent the conditions in Figure 3a. The first index would represent the `HuntLocation` ontology and the second would represent the range of possible times available to the `Time` context. The message would be stored at the appropriate points in both dimensions. Note that the `Time` conditions express a range of times (**:00 $\leq T \leq$ **:01), meaning that placement along the `Time` dimension would need to be expressed as regions instead of points.

6. CONCLUSIONS

We have described the MediateSpace middleware which offers decentralised information distribution with context based access control. Also supported are a number of context and pattern-based structures which operate on received data. These are mediated context sharing, triggers for dynamically executing actions, filters to restrict unwanted data and module handlers to process the data. System structures and state can be dynamically modified through the context scripting and state management capabilities.

7. FUTURE WORK

There are several aspects of design which require further investigation:

- The incorporation of an OWL reasoning engine into the regional nodes to handle condition evaluation,
- Stronger enforcement of privacy and access control through the inclusion of encryption and group membership,
- A precise Regional Node election protocol with analysis to ensure that the network structure does not result in bottlenecks,
- A robust replication and consistency scheme for both the distributed data structure and messages,
- A garbage collection mechanism for purging expired or unnecessary messages.
- A simple methodology for incorporating legacy context sources and sinks with MediateSpace.

In the longer term, we intend to complete the MediateSpace implementation and to test it's performance and utility through simulation.

8. ACKNOWLEDGEMENTS

This work was supported by the Engineering and Physical Sciences Research Council, grant EP/F064330/1. We also wish to thank Simon Fleming and the anonymous reviewers for their constructive feedback.

References

[1] Katie Shilton. Four billion little brothers?: privacy, mobile phones, and ubiquitous data collection. *Commun. ACM*, 52(11):48–53, 2009.

[2] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[3] Amy Murphy Dept and Amy L. Murphy. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 524–, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Rajiv Ranjan, Aaron Harwood, and Rajkumar Buyya. Peer-to-Peer Tuple Space: A novel protocol for coordinated resource provisioning. Technical report, The University of Melbourne, Victoria, Australia, 2007.

[5] Gianpaolo Cugola, Gian P. Picco, and Politecnico Di Milano. PeerWare: Core middleware support for peer-to-peer and mobile systems, 2001.

[6] C. Archer. *Process Coordination and Ubiquitous Computing*, chapter 1, pages 11–29. CRC Press, Inc., 2002.

[7] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. Mobile data collection in sensor networks: The TinyLIME middleware. *Pervasive and Mobile Computing*, 1(4):446 – 469, 2005. Special Issue on PerCom 2005.

[8] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM.

[9] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *Knowl. Eng. Rev.*, 18(3):197–207, May 2004.

[10] Panu Korpipaa, Jani Mantyjarvi, Juha Kela, Heikki Keranen, and Esko-Juhani Malm. Managing context information in mobile devices. *IEEE Pervasive Computing*, 2(3):42–51, 2003.

[11] Tao Gu, Hung K. Pung, and Da Q. Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18, January 2005.

[12] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann, and Werner Retschitzegger. Context-awareness on mobile devices - the Hydrogen approach. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9*, page 292.1, Washington, DC, USA, 2003. IEEE Computer Society.

[13] J. Santa and A. F. Gómez-Skarmeta. Sharing context-aware road and safety information. *Pervasive Computing, IEEE*, 8(3):58–65, July 2009.

[14] Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, Ronald A. Peterson, Gahng seop Ahn, and Andrew T. Campbell. Metrosense project: People-centric sensing at scale. In *WSW 2006 at Sensys*, 2006.

[15] Amy L. Murphy and Gian Pietro Picco. Using LIME to Support Replication for Availability in Mobile Ad Hoc Networks. In *In Proceedings of the 8th International Conference on Coordination Models and Languages (COORDINATION 2006). Number 4038 in Lecture Notes on Computer Science*. Springer, 2006.

[16] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, volume 14, pages 47–57, New York, NY, USA, June 1984. ACM.

[17] Stefan Berchtold, Daniel A. Keim, and Hans P. Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[18] King I. Lin, H. V. Jagadish, and Christos Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal: Very Large Data Bases*, 3(4):517–542, 1994.